IMPROVING AND EXTENDING A 2D OCTOMAP BASED SLAM RAY-CASTING SYSTEM FOR 3D *

Henrique Ribeiro, João Lucas, João Costa, Tiago Duarte
Student
FEUP
Porto
{up201806529, up201806436, up201806560, up201806546}@up.pt

ABSTRACT

The goal of SLAM algorithms (Simultaneous Localization and Mapping) is to map an environment navigated by an autonomous vehicle, while simultaneously locating it in the map, without access to pre-existing charts or external devices. This paper will detail the design and implementation of a system that focuses on the mapping a 3D environment. This is based on previous work by the authors, bringing improvements (both in terms of features and in terms of performance) to the area of edge/object detection, and map building.

***K*eywords** SLAM, octomap, sonar, AUV, ray-cast, dda, bresenham, computer vision, Canny

1 Introduction

The development of this project was proposed by the CRAS lab at FEUP, presenting a problem of mapping the bottom of the ocean using a *SHAD AUV*. This *AUV* collects information of its environment using a sonar.
The *AUV* will navigate underwater environments. This presents some challenges associated with the environment, such as: the lack of wireless communication between the *AUV* and the main land, the diverse types of noise in the collected data, and the limited computation capabilities of the device. It is of interest to continue exploring the previous work on mapping 2D environments by the authors in order to expand it to 3D, improve the results of the edge/object detection, and optimize the ray-casting operations.
To achieve these goals, the team resorted to implement a hash table/unordered set for the ray-casts and use Canny's algorithm for image processing.

2 Ray-casts improvements

The team noticed that the set operations needed for the ray-casts operations in point-cloud updates, namely the set merge operation, were responsible for about 50% of the time used. To improve this, the team developed a custom hash table implementation to replace C++ unordered set.

C++ unordered set implementation isn't fit for our problem for two reasons. Firstly, its hash table uses closed hashing. This leads to increased memory usage and less performance overall. Secondly, C++'s hash table follows a general purpose design. As such, the implementation is based on pointers to objects. In the ray-casting use case, the objects stored in the set are smaller than a pointer in 64-bit systems ($48bit < 64bit$). Storing the raw objects instead of a pointer leads to reduced memory consumption. It also prevents a pointer following operation, which is expensive.

2.1 Hash Sets Details

To use a hash set, the hash for each element to be stored must be calculated. The hash table uses this hash to calculate the index where the element will be stored. To do this, the hash table uses the modulo operator on the hash value and

* *Citation*: **Henrique R. et al. Improving and extending a 2D Octomap based SLAM ray-casting system for 3D**

the current hash table size. Since this implementation uses open-hashing, it is necessary to keep information relating to deleted entries: tombstones. Upon deleting an entry, the hash table places a marker in its place (same index) indicating there was an entry in that position, so probing isn't broken. This ensures that when looking up elements, the search doesn't stop prematurely.

2.2 Hash Collision Resolution

When calculating the index of an element to insert, there are cases where another element already occupies that index: collision. There are multiple collision resolution technics/algorithms available for these situations. These technics calculate a new index to insert the element that caused the collision. The team implemented and analyzed three different open-addressing resolution technics, in order to find the one that gives the best performance for the use case described.

2.2.1 Linear Probing

With linear resolution, when two elements end up in the same index, the technic tests the next index, repeating this process until it finds a free index. This method is very simple to implement as well as verify, but it is not, however, the most efficient algorithm, because it causes dense collision areas. The team used this method to establish a baseline for the other methods.

2.2.2 Quadratic Probing

Quadratic resolution is a technic that probes the $StartingIndex + i^2$th index in the ith iteration, whenever there is a collision in the previous iteration. This is an attempt of reducing the density of collision areas.

A big problem of quadratic probing is that it can lead to cycles. This means that there had to be some changes to the algorithm to guarantee that a free slot is always found. The size of the table was the first characteristic to change. The size needed to be a power of 2 to guarantee that there would be no cycles. On top of that, there needed to be some changes done to the expression used to calculate the next index to probe. Instead of using i^2 in the expression, the quadratic probing offset became $(i^2 + i)/2$ [1].

2.2.3 Double Hashing

Double hashing takes advantage of two different hashing functions. The first one calculates the initial index to place the element, and the second calculates the offset for the index in the case of collision. The combined hash function thus becomes $H(e) = H1(e) + i * H2(e)$, where i is the ith iteration of the hashing function. The use of the second hashing function results in a non-linear addressing method of the hash set, causing fewer collisions and reducing the density of collision areas.

Similarly to the quadratic probing technique (see section 2.2.2), double hashing can get stuck on cycles. To solve this problem, it is enough to guarantee that the result of the offset function (in this case $i * H2$) and the table size are relatively prime. This can be achieved by making one of the numbers prime, i.e., the table size. However, this would add a significant overhead to the set resizing operation, since searching for the next prime number is an expensive operation. Furthermore, the performance of the modulo operator would be negatively affected by making the denominator a prime number. The group chose an alternative approach: ensure the hash table size is always a power of two, and guarantee that the offset always return an odd number [2]. The power of two part is straight forward. The group made the second hash function always return an odd number by always setting its least significant bit [2].

2.3 Dynamic resizing

When the set reaches a certain load factor, it is necessary to expand the number of elements it can hold. Resizing is done by expanding the allocated size of the vector to the double of its current value. This keeps the value a power of two, because the initial size of the container is a power of two. After this resize, every element previously stored that was not deleted, needs to be inserted again, as the function used to calculate the index where each element is place changed. Essentially, this creates a new hash table with double the capacity. This operation, clears the tombstones from the container.

The hash table caches the hash value of each element inserted into the set. This is done both to make lookups faster and to make the re-insert part of the resize operation faster.

It should be noted that the resize operation produces a significant overhead. To mitigate this overhead, the initial set size should be as close as possible to the final number of inserted elements, thus minimizing the number of resizes done.

2.3.1 In-place dynamic resizing

The resize operation as described in the previous chapter consumes a 50% more memory than necessary in order to make the resize operation as simple as possible. The group implemented a second dynamic resizing method that resizes the container in-place, doubling its size instead of allocating a new container. To do this, the algorithm doubles of the set and then iterates over its first half inspecting all non-tombstone elements. Then, the algorithm calculates the new index of each element. If the index falls on the second half of the container, it is placed there as the usual insert operation mandates. If the index fell on the first half of the container, it is placed on a temporary buffer. After all elements in the first half are processed, the algorithm cleans this part of the container and inserts all elements in the temporary into the set, as usual.

2.4 Results & Benchmarks

2.4.1 Comparing Different Collision Resolution Technics

As can be seen by image 1, when performing 4 million inserts, the collision resolution technic with the best results is the **quadratic probing**.

The *spikes* in the graphs correspond to the moments resize operations happened. As discussed before, these resize operations come with a significant overhead, which causes the loss in the performance.

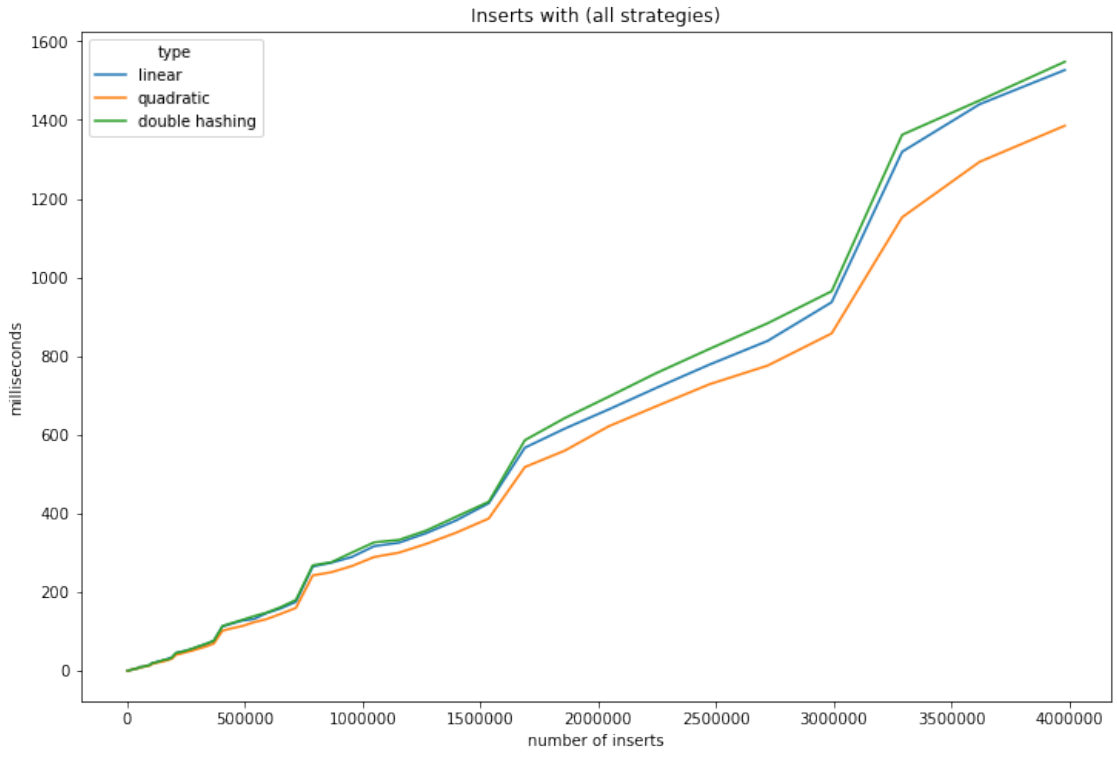

Figure 1: Comparing Insert Times Using Different Collision Resolution Technics

2.4.2 Impact of Resizing

With the results from the section 2.4.1 it became clear that resizing had a bigger overhead the more elements the set had. Additionally, the smaller the starter set the more resizes have to be done, leading to a bigger overhead.

Looking at image 2, it's possible to see the impact resizes have. To avoid this, it is essential to initialize the set with a number of elements as close as possible to the actual amount it will have.

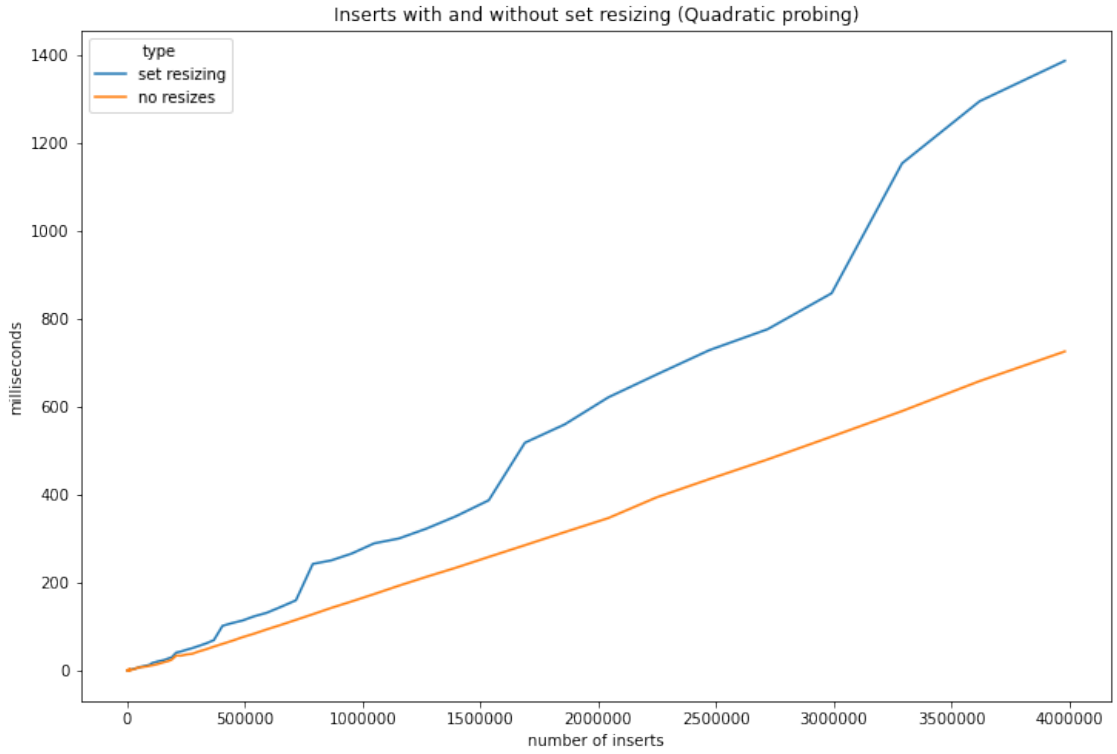

Figure 2: Comparing Insert Times With and Without Resizes

2.4.3 Comparing Number of Collisions

As seen in figure 3, linear probing has the highest number of collisions, while the other two technics have around the same amount of collisions.

Even if the number of collisions are similar, the quadratic probing technic has a better performance as it does not have the overhead of having to calculate new hashes every time there is a collision. The group believes that caching the value of the second hash function of the double hashing technique could lead to different results.

2.4.4 Merge – Impact of resizing

The merge operation consists of iterating through one set and inserting all of its elements into another. This operation can lead to resizes, especially if the set that is being merged into is smaller than the source set.

To mitigate the overhead of the resizes, the group implemented a pessimistic heuristic: assuming that every element in both sets are different, it is possible to calculate the necessary space needed for both sets and perform only one initial resize (if needed). This proved to be beneficial for performance in synthetic test, as can be seen in figure 4.

2.4.5 Dynamic resizing methods comparison

As described in previous sections, the team implemented two dynamic resizing methods: all-at-once resizing, which allocates a new hash table, and an in-place resizing, which expands the current container. The second method is more complex, but uses significantly less memory. Fig. 5 corroborates the added complexity: the in-place dynamic resizing method takes slightly longer to perform.

Given that the performance difference isn't too big, but the difference in memory usage is, the team decided that the in-place dynamic resizing should be the default method used.

2.4.6 Comparing With C++ unordered_set

The goal of this set is for it to be faster than the standard library's unordered_set. For this reason, it is important to compare the two in the real dataset.

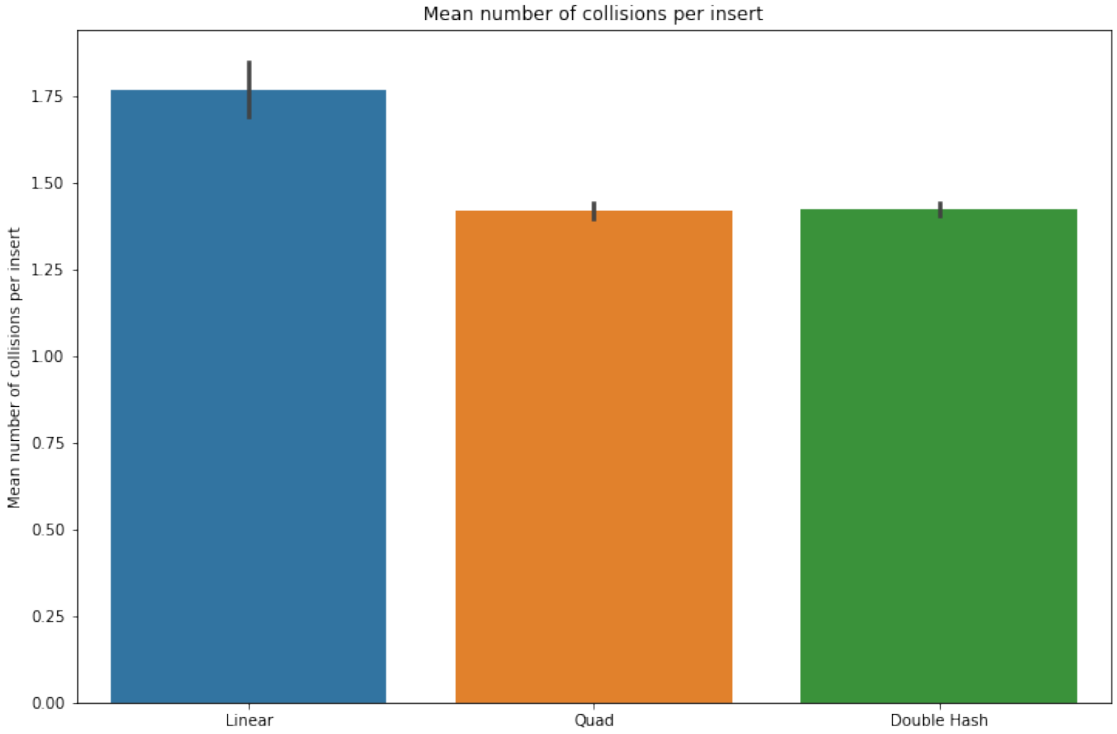

Figure 3: Comparing the Number of Collisions Between all Technics

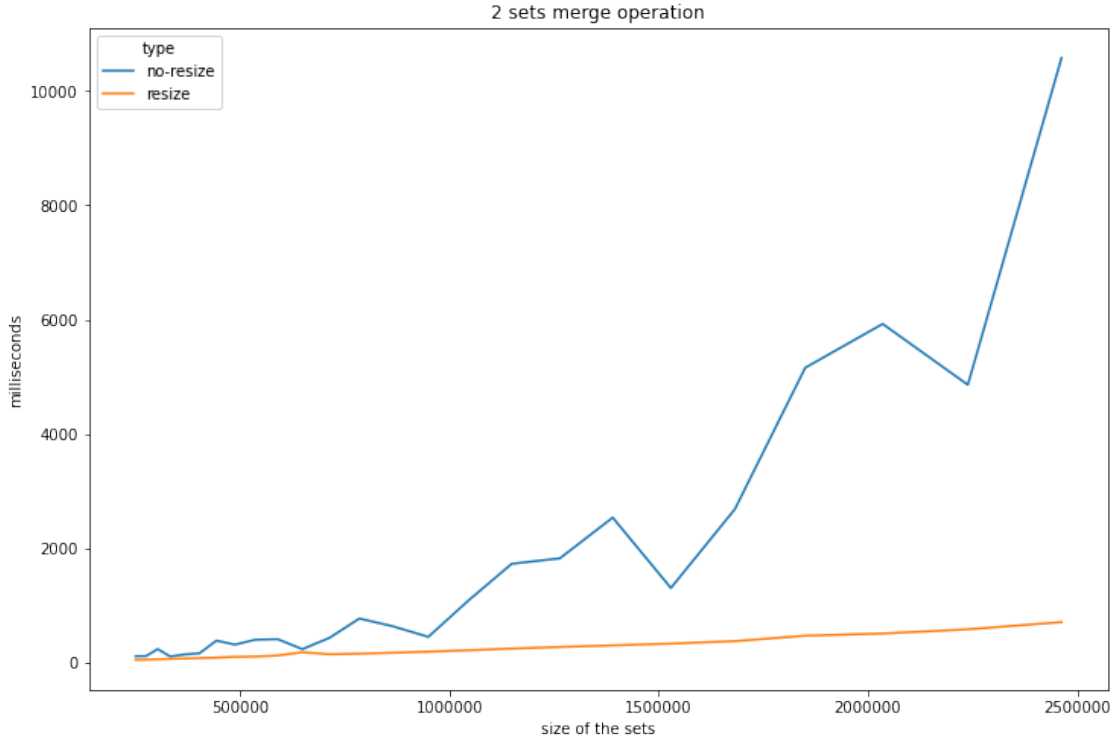

Figure 4: Comparing Pessimistic Resizing and Normal Resizing During Merge

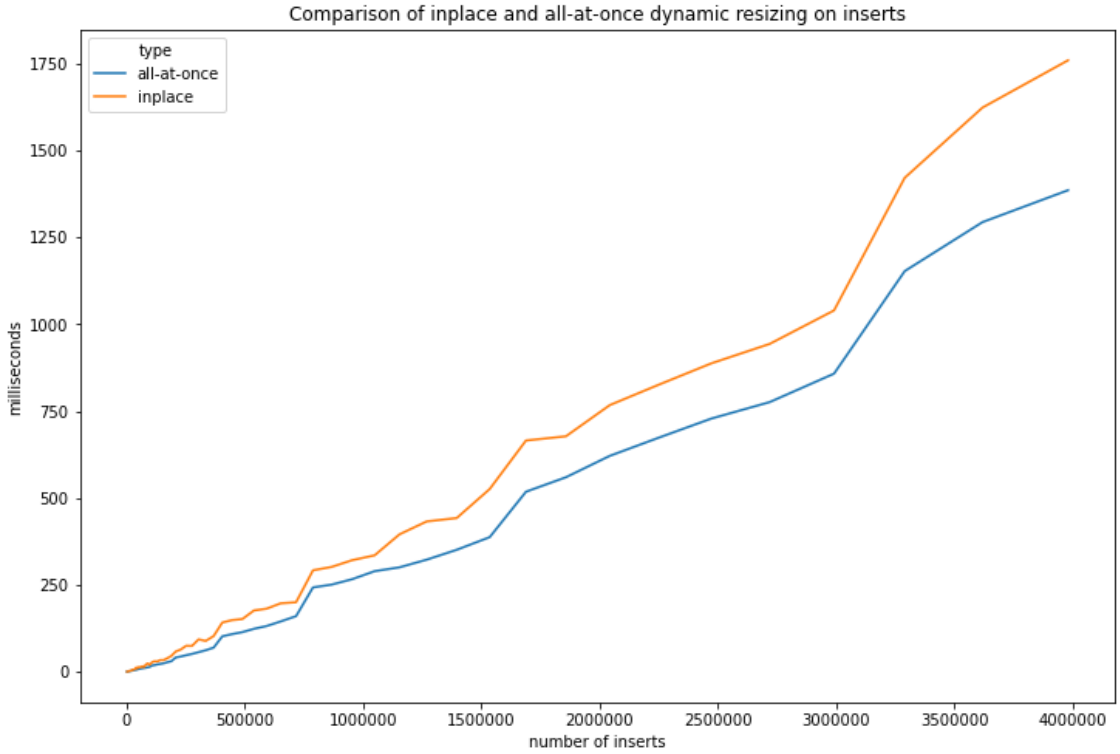

Figure 5: Comparing dynamic resizing methods

Figure 6: Comparing with C++ unordered_set Without Resize Optimizations

Figure 7: Comparing with C++ unordered_set With Resize Optimizations

As can be seen in figures 6 and 7, the performance of the implemented sets is worse than the C++ unordered_set if not using the resize optimization, but better if it is. The group believes this is because C++ sets don't need to be resized.

Table 1: Time taken to run pipeline on real dataset for different set implementations

	C++ sets	Our sets	In-place resize
Time taken	$\approx 23s$	$\approx$ 13s	$\approx$ 10s

By running tests on the real pipeline, using a real dataset, the team obtained the results expressed on Tab. 1. The times expressed in the table indicate that the new set implementation performs better for the project use case than C++'s unordered_set. Furthermore, it shows that, even though the sizing operation is slightly slower for the in-place resizing method, the gains in memory lead to better temporal performance overall.

3 3D Mapping

The team's previous implementation, described in [3], can map an environment in two-dimensional space by estimating cells covered by the sonar's beams in the horizontal (Oxy) plane. This section serves to discuss the approach that was adopted in order to transition from 2D mapping to 3D mapping.

The only step that requires adaptation is the cell estimation step, since the developed octomap already supports 3 dimensions and all other sonar related operations (preprocessing, probability estimation, etc...) are independent of the map's dimensions. This implies that, for each beam, rather than estimating an area similar to a cone, a three-dimensional volume needs to be covered.

3.1 Approach

Similar to the 2D solution, each beam's volume will be estimated using ray-casts. The approach as is defined in [4] uses one horizontal and two vertical divisions. A representation of these rays can be found in 8.

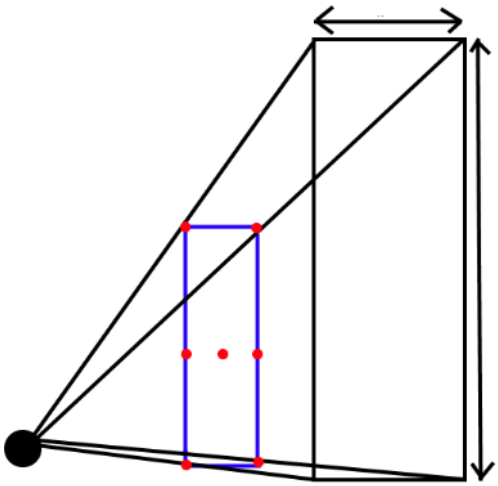

Figure 8: Volume Estimation using Raycasts

This fails to map all visible cells, which was opted for due to performance reasons: “This can be expanded to cover most of the volume but that would drastically increase the operating time.“[4]. The team, however, has found that it can use more divisions without hindering performance, due to previously discussed optimization approaches, such as parallelization and the custom hash set implementation, that made ray-casting much faster.

The size and type of the estimated volume depend on the sonar’s characteristics. The sonar used for the purposes of this project, *SHAD AUV*, spans each of its beam 3º horizontally and 35º vertically. This needs to be taken into account when deciding on the number of divisions in both dimensions, as, in order to capture the same amount of precision, the vertical plane needs to be more divided than the horizontal plane.

3.2 Results

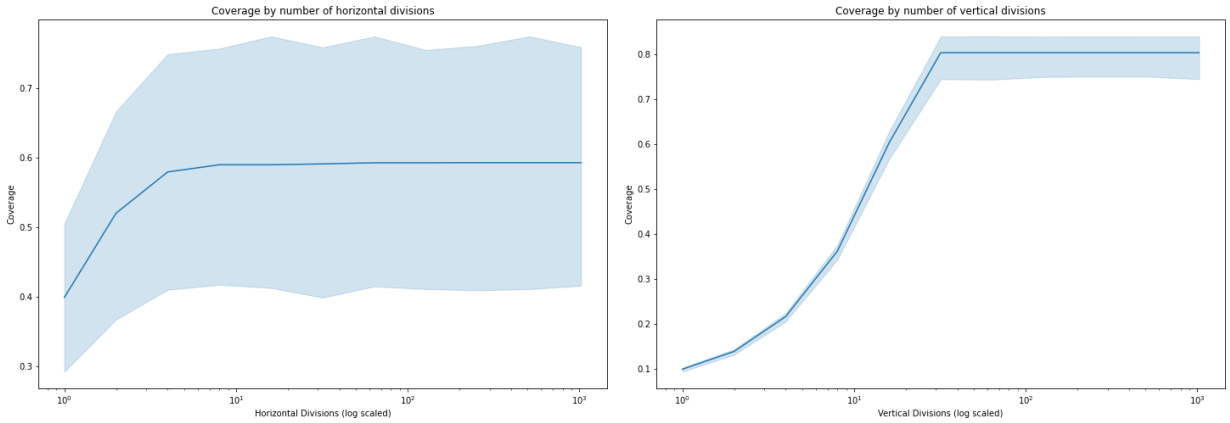

Figure 9: Cell Coverage by Number of Divisions

For this reason, in order to find the optimal number of divisions for both planes, the team has measured cell coverage for each plane. The results for these experiments can be found in 9, 10. Cell coverage increases linearly for both planes until it reaches the optimal number of divisions. The found optimal number of divisions is 64 and 32, for the vertical and horizontal plane, respectively. Time also seems to increase linearly, which was not expected, as the developed ray-cast point cloud implementation is $O(N^2)$. The team believes that this is the case due to a high number of overlapping cells between all beams, especially at the begging of each ray. This, coupled with fast merging set operations and paired with parallelization, leads to a much reduced execution time than expected.

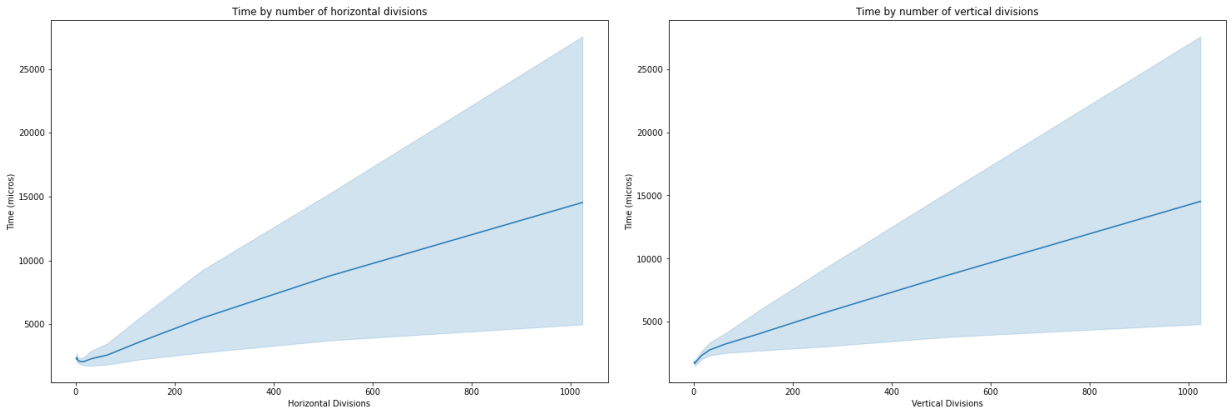

Figure 10: Time by Number of Divisions

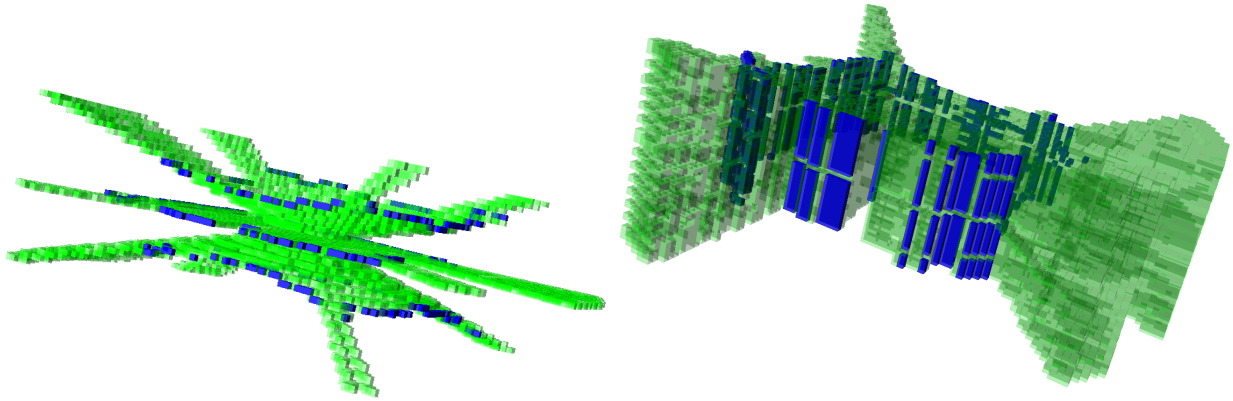

Figure 11: Octovis representation using 2 divisions (left) and optimal divisions (right)

Figure 11 also showcase the generated map for a low number of divisions (2 horizontal and vertical), and the optimal number of divisions

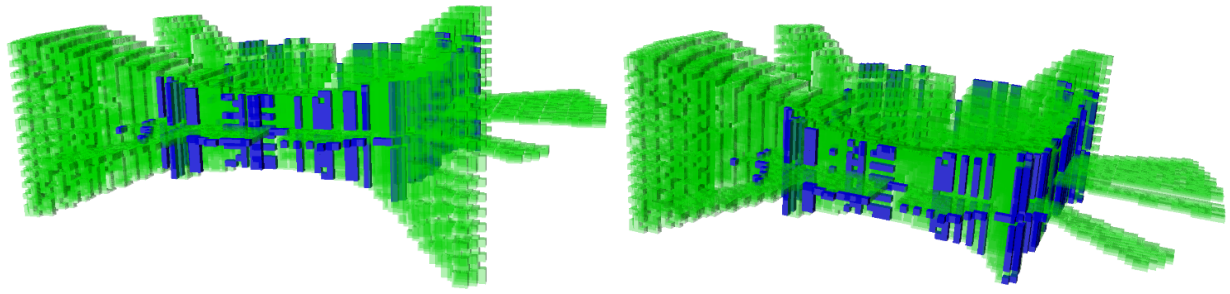

Figure 12: Octovis representation after 1 sweep (left) and after 27 sweeps (right)

In addition, figure 12 highlights incremental probability updates by showing a map representation using only the first scan and all 27 scans of the dataset. As can be seen, the system is able to be more sure of cells that are occupied when it was exposed to them repeatedly.

4 Object Detection

Accurately detecting obstacles in a beam is an essential step to map the environment precisely. As such, free, occupied, and unknown cells need to be properly identified. A subset of measurements near the sonar's source are self-reflections of the vehicle's body, and are always discarded.

As described in [3], the simplest approach to determine an object is to use a simple threshold [4] to determine when the difference between two sequential measures in a beam is significant enough for the area to be considered an edge. The value of this threshold need to be defined a priori.

In the hopes of improving the results, the team looked for alternatives, namely, an edge detector, which should abide by three main criteria: Good detection, as it should accurately detect as many of the edges present as possible, with a low error rate. Good localization, meaning that points should be localized at the centre of the edge. Single response, as each edge should only be marked once, and edges created from noise minimized.

As a disambiguation, while an edge detector serves as a base for computing contours or shapes, it only detects edges in an image. Applied to our sonar images, it means it won't implicitly detect the tank walls, their width, or their centre, only the points where the wall most likely starts.

One of the most prominent algorithms used is the Canny Edge Detector[5], which proposed all the criteria mentioned above. As such, after some promising results with OpenCV's library[6], the team attempted our own implementation, in hopes some parts could be adapted to better fit our sonar images.

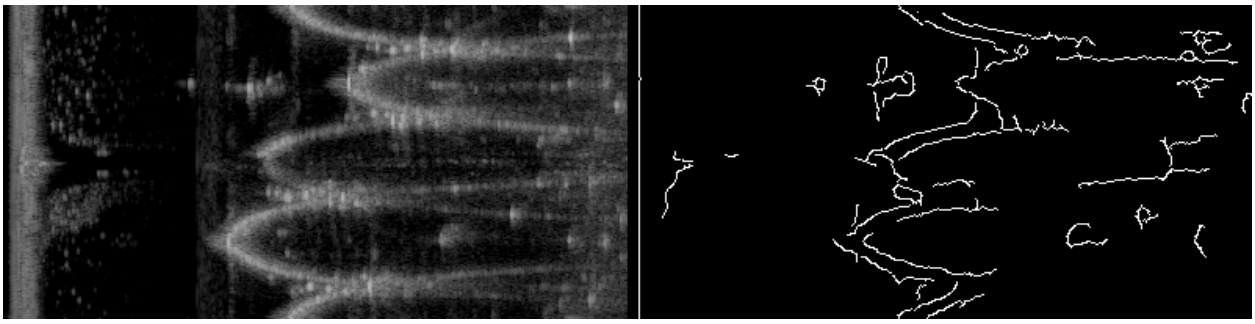

Figure 13: OpenCV Canny Results

4.1 Canny's Edge Detector

Canny's Edge Detector is best described as a sequence of operations rather than an algorithm itself, and consists of four phases: image smoothing, differentiation, non-maximum suppression, and a double threshold with hysteresis. The renowned Lenna image 14 will be used to demonstrate these various steps.

Smoothing and Differentiation

The first step aims to reduce the noise present in the image by applying a Gaussian kernel, usually with size of 5, with a configurable standard deviation σ.

The differentiation step calculates the first derivative in both the horizontal and vertical directions, applying a convolution with an edge detector operator such as Robert's Cross, Prewitt's Operator, Sobel's Operator, or Shcarr's Operator. To accurately represent the gradient, however, its intensity and angle need to be calculated using the following equations.

$$G = \sqrt{G_x^2 + G_y^2} \qquad \Theta = atan2(G_y, G_x)$$

In our implementation of Canny, the team started by using Scharr's operator, as it represents a more accurate derivative for a $3x3$ kernel. As the results were mostly indistinguishable from Sobel's operator, it was changed to the latter. This second kernel has a higher weight in the centre pixels than the perfect mathematical equation, which is beneficial for the scan, as a horizontal line represent measurements from the same scan.

As an optimization, to avoid iterating through the entire matrix and computing the square root operation, the thresholds are multiplied by themselves instead. The benefits of this change are more apparent when processing larger images, such as Ultra HD 4k images.

Figure 14: Original

Figure 15: Differentiation

Non-Maximum Suppression

The third step is the non-maximal suppression, which aims to fulfil the single response and good locality criteria, by removing all pixels that are not the local maxima of its respective edge. The simplest approach to solve this issue is to reduce all angles into 8 sectors16. Afterwards, the pixel is compared to the neighbours in its respective direction. As an example, when processing a pixel in sector 1, it is compared to the pixels above and below.

Figure 16: Non-Maximum Suppression Directions

This simpler tactic is not without flaws, however, as there is some ambiguity in determining which sector a pixel sitting very close to the boundary belongs to. As a consequence, there are cases where a pixel vanishes, because it was considered as a part of the wrong sector, where an edge should have been preserved. This occurs in our implementation, and while there is no concrete evidence, the team suspects the same also applies to OpenCV.

A more accurate but computationally complex method is to interpolate the values of the pixels in ambiguous cases, so instead of directly comparing a pixel with two others, its angle could be used to weight the pair of pixels of both ambiguous sectors. While this could likely jeopardize the single response criteria, it could improve the accuracy of our use case, ensuring the tank's walls would not have the missing pixels it currently has.

Double Threshold and Hysteresis

The use of a single threshold usually causes edge lines to appear broken, due to edge values fluctuating above and below the threshold, a phenomenon called streaking. For object detection in SLAM, this is unacceptable, as the tank walls randomly phasing in and out would result in an inaccurate *octomap*. To avoid this behavior, two thresholds and hysteresis can be employed to classify edge pixels in three categories, as can be seen in figure 20.

- **Strong edge**: Pixels whose Value is above the upper threshold, and are always considered an edge (green line).
- **Weak edge**: Pixels whose value rests between the lower and upper threshold, which are only considered an edge if they belong to an edge that contains at least one strong pixel. The algorithm keeps weak pixels connected to strong pixels (blue line), but discards weak pixels without any connection (red line).
- **Not an edge**: Pixels whose value is below the lower threshold, and are always discarded (pink line).

Figure 17: Non-Max Suppression

Figure 18: Double Threshold

Figure 19: Hysteresis

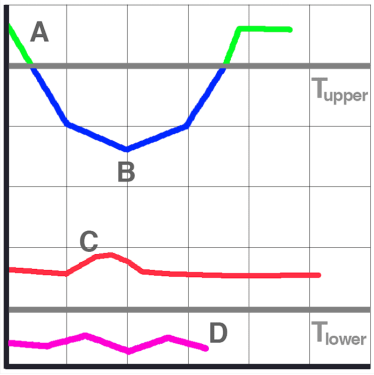

Figure 20: Hysteresis

The use of hysteresis, however, demands the use of an algorithm to determine which pixels belong to which edge. For this, the family of algorithms called Connected Component Labelling, or, sometimes, Blob Analysis, can be used.

4.2 Connected Component Labelling

The goal is to label each connected region in a binary image, often called blobs, with the same unique value. There are various algorithms to solve this, with differing time complexities, but only some will be mentioned, namely the one component at a time algorithm and the two-pass algorithm. During the research, some papers mentioned linear time implementations that could be parallelized[7], however, due to time constraints, the team was unable to explore them.

There are also two possible options for these algorithms, either four-way or eight-way connectivity. This definition of connectivity determines which pixels are considered as being part of the neighbourhood, either the horizontal and vertical pixels (four-way) or all cells surrounding the pixel (eight-way).

The aforementioned algorithms will be detailed below, alongside the adaptations required to better suit the hysteresis problem.

4.2.1 One Component at a Time

The idea behind this algorithm is to iterate through the entire edge whenever a new unlabelled pixel is found. The time complexity for this algorithm is unknown, but is at least $O(N^2)$, as it does a single pass through the image, but the number of extra pixels it has to iterate through depends on the complexity of the image.

As such, the algorithm iterates through every pixel, and keeps an auxiliary queue to save the pixels that are candidates to belonging to the same edge. If a pixel is strong and unlabelled, then it is given the current label and added to the queue. Then, another loop is used to empty the auxiliary pixel queue.

While the queue is not empty, pop an element, and if it is above the lower threshold and unlabelled, give it the current label and add its neighbours to the queue. Otherwise, simply skip the pixel and keep analysing the queue's elements.

When the queue has no elements remaining, finally increase the label value being attributed. When applied to the current use case, this label needs not be increased, as by setting the final image's pixel value to 255, the output binary image is already created.

Some literature describes adaptations to add parallelism to this algorithm, but this possibility was not explored.

4.2.2 Two-pass Algorithm

The two-pass algorithm, as the name implies, only passes through the entire image twice, and uses an auxiliary equivalence table. As such, it can be divided into two main steps, having a time complexity of $O(2N^2)$, plus the complexity of the structure used to implement said equivalence table. This process requires another image array to be created, capable of storing 32-bit integers instead of 8-bit, as an image can have many labels. This extra buffer initially has all labels set to 0.

The first step iterates through each pixel and analyses its neighbours' labels, and depending on the labels found, different actions are taken:

- **No non-zero neighbours:** Create a new label, as it must be a new component.
- **One non-zero neighbour:** Give the same label it found.
- **Many non-zero neighbours with the same label:** Give the same label it found.
- **Many non-zero neighbours with the different labels:** Set to the lowest value label and add a new entry to the equivalence table, to mark the various labels found as being part of the same component.

An optimization to this step is to only analyse half of the neighbours, halving the number of visited nodes. As such, in 4-way connectivity only the left and upper pixels are scanned, whilst in 8-way connectivity only the neighbours from the left pixel to the upper right pixel are scanned. This still ensures the correct solution due to the remaining pixels always being set to 0, as they are not processed at that stage.

The second pass simply iterates through each pixel, checking the equivalence table for the lowest label possible and updating its value.

The resulting image has no method to automatically discern which edges are strong however, so the team proposes an adaptation to the algorithm to avoid a third pass, based on the concept of coupling meaning to the value of a label.

Proposed Changes

The original algorithm's labels start from 1 and are incremented for each new possible component. By having two sets of labels, one incrementing from 1 and another decrementing from an arbitrary maximum value. Low labels correspond to strong edges, whilst high value labels correspond to weak edges. This ensures that if a pixel's equivalent label is a low value, it will belong to a strong edge.

Applying the current rule set could result in edges that started as weak never being marked as strong. As such, when processing a strong pixel, instead of directly copying the label from its neighbours, if the label to be assigned was weak, a new strong label is created, and the equivalence table updated with the new relevant entries.

Unlike the one component at a time algorithm, the format of this output is different from the final image, so a conversion is required. This can be done during the second pass, as by the moment the lowest value label is found, it is possible to determine if the pixels belongs to a strong edge (a component whose value is below the highest strong label assigned).

With the algorithm fully described, an implementation for the equivalence table is required. The first option is to use a HashMap. However, using this data structure would require extra processing before the second pass, in order to invert the entries and ensure they are mapped to the lowest value label. On a closer look, the operations required can be easily recreated using Disjoint Sets.

Disjoint Sets

Initially, the intention was to use Boost's disjoint sets implementation, but due to poor documentation and the class being moved to the pending package, it was ultimately decided against it.

As such, a basic implementation was created, but its performance wasn't satisfactory. By adding path compression on find, 60ms were saved when processing the sonar scan. Additionally, by altering the union operation to always prioritize keeping the smallest node as the root, as this would ensure the trees are always compressed, it is possible to easily identify if a set (edge) contains a strong pixel or not. This, however, makes it impossible to use the union by rank or union by size optimizations.

Another compromise of using disjoint sets is they are implemented using simple C arrays, and the number of labels must be determined a priori. This prevents the weak labels from starting from MAXINT, and instead from the maximum size of the disjoint sets. Even without the changes to the two-pass algorithm, no data regarding the maximum number of labels on any given image was found.

The number of labels also depends on whether 4-way or 8-way connectivity is being used, with the former having the most possible labels. As an approximation, the block with the dense number of labels was derived, and later used as a pessimistic approach for the number of possible labels. Therefore, the size of the array for the disjoint sets was a third of the image size.

4.3 Results

When comparing the output of our implementation 21 to OpenCV's 13, the lower noise level was not achieved, and the most important edges, the ones outlining the tank's walls, are not as well-defined. This issue could be solved by more parameter tuning, but the team decided against further exploring this issue after analysing various performance benchmarks.

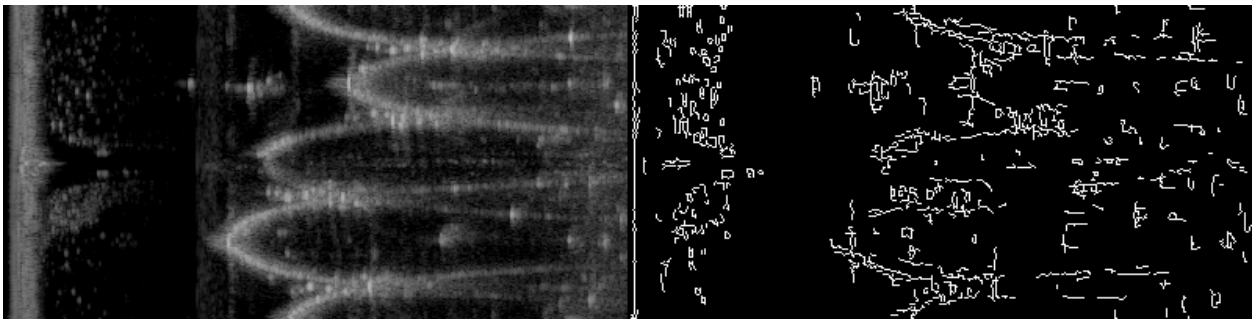

Figure 21: Custom Canny Implementation

To test the performance of our implementation, the algorithms were tested on a 399x200 image of the sonar and a 4k image22. The results obtained were below expectations, but some interesting conclusions can be reached when comparing the time complexity of all the Canny edge detectors.

Compared to OpenCV's Canny, our implementation is 15 times slower on the sonar image, with the one component at a time algorithm being slightly slower. When processing a much larger 4k image, although the two-pass algorithm's fixed time complexity kept it 15 times slower than OpenCV, the one component a time algorithm's performance becomes 30 times worse than OpenCV's. This is due to its complexity scaling with the number of pixels analysed, so a larger image with lengthier edges causes performance to worsen significantly.

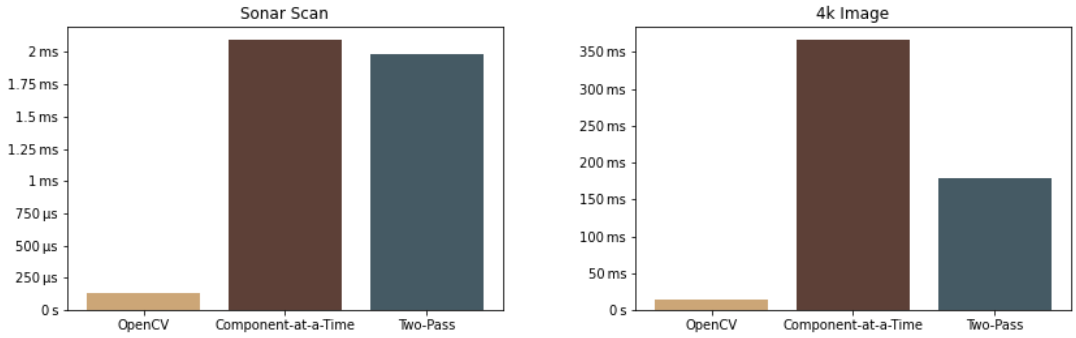

Figure 22: Canny Performance

Faced with these results, the team decided to abandon our implementation and use OpenCV's, as too many optimizations and parameter tuning would be required to replicate the results and performance achieved with OpenCV.

5 Future work

Based on the results found, the group identified several aspects where future work should be focused. Of these, the main focus of future work should be on solving the edge continuity problem, and exploring the *AUV* location part of SLAM.

When it comes to the newly implemented sets, it might be worth to explore different collision resolution techniques, such as Cuckoo hashing. Furthermore, the group should explore the findings of Daniel Lemire, et al. in "Faster Remainder by Direct Computation: Applications to Compilers and Software Libraries" [8], where they describe how to calculate the modulo operator, which is one of the most expensive operations in Hash tables, faster.

Regarding the edge detector to be used for object detection, the plan is to continue using OpenCV's implementation. Nonetheless, the group intends to explore options to automatically determine good parameters, to achieve more resilient results, starting by exploring the zero parameter Canny proposed by Adrian Rosebrock[9].

6 Conclusion

In this paper, the team described the steps taken towards expanding their previous mapping system to 3D. Furthermore, the existing system was improved, with special attention when it comes to the performance of the ray-cast operations and the edge detection.

In terms of set performance, the team managed to reduce the time taken to run the pipeline by up to 50%, depending how on the size of the dataset. This result is a major improvement over the previous version of the pipeline. This improvement serves as support for the move to 3D of the pipeline, which performs many more ray-cast operations for the same dataset.

Acknowledgments

This research was partially supported by Bruno Ferreira and António José from the CRAS laboratory at FEUP. The team would also like to thank João Fula for his research paper in the topic, which assisted us in our own.

Once again, the team would like to thank António José for his availability to assist us and insights, both from his thesis and from his comments/ideas/tips during the development of the project.

References

[1] Virginia Tech. Hashing tutorial: Section 6.3 - quadratic probing. `https://research.cs.vt.edu/AVresearch/hashing/quadratic.php`.

[2] University of Waterloo Carlos Moreno. Hash tables - double hashing. `https://ece.uwaterloo.ca/~cmoreno/ece250/2012-02-01--hash_tables.pdf`.

[3] Henrique R. et al. Improving and extending a 2d octomap based slam ray-casting system for 3d. May 2022.

[4] João Pedro Bastos Fula. Underwater mapping using a sonar. *IEEE*, 2020.

[5] John F. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8:679–698, 1986.

[6] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[7] Kenji Suzuki, Isao Horiba, and Noboru Sugie. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding*, 89(1):1–23, 2003.

[8] Daniel Lemire et al. Faster remainder by direct computation: Applications to compilers and software libraries. 2019.

[9] Adrian Rosenbrock. Zero-parameter, automatic canny edge detection with python and opencv.